Jan 31, 2022

# TEMPORARY MEMORY ALLOCATOR [C/C++]

While programming you often need some memory to do something but you don't know at compile-time how much you need. Instead you have to use **malloc** from C or **new** from C++ (or more commonly **std::vector**) to request a certain amount of memory to be allocated for you on the heap. That's just how it works.

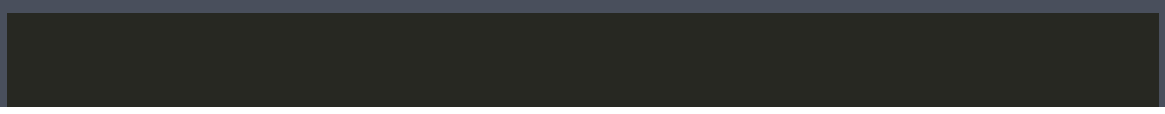
You might not question this sort of pattern while doing normal programming but once you start getting into game development you'll quickly learn these sorts of dynamic memory allocations that occur at runtime can have performance implications.

There is an interesting alternative to some dynamic memory allocation that I want to discuss: a temporary memory allocator.


- **Examples of Temporary Heap Usage**
- **Some Problems with the Heap**
- **Alternative #1: Static Arrays**
- **Alternative #2: Temporary Memory Allocator**
  - **Stepping Through**
  - **The Stack**
  - **Managing Scope**
  - **A Global Object**
  - **In Practice**
- **Conclusion**
- **Further Reading**


## Examples of Temporary Heap Usage

Consider the following pattern when working with the Vulkan API:



```
// C
uint32_t layer_count = 0;
vkEnumerateInstanceLayerProperties(&layer_count, NU
VkLayerProperties* layer_properties = malloc(sizeof
vkEnumerateInstanceLayerProperties(&layer_count, la

// C++
uint32_t layer_count = 0;
vkEnumerateInstanceLayerProperties(&layer_count, NU
VkLayerProperties* layer_properties = new VkLayerPr
vkEnumerateInstanceLayerProperties(&layer_count, la

// C++ (alternative)
uint32_t layer_count = 0;
vkEnumerateInstanceLayerProperties(&layer_count, NU
std::vector<VkLayerProperties> layer_properties(lay
vkEnumerateInstanceLayerProperties(&layer_count, la
```

This pattern repeats itself many times. You first query to see how many of something there are, you allocate some memory to hold the objects, and then you fill the memory.

Another example is when you need to load a file into memory so that you can then process it into a different form at which point you no longer need the memory. Something like a GPU shader or a model loaded from an OBJ file.

```
FILE* fp = fopen(path, "r");

fseek(fp, 0, SEEK_END);
size_t file_size = ftell(fp);
rewind(fp);

char* buf = malloc(file_size);
fread(buf, sizeof(*buf), file_size, fp);
fclose(fp);

// Create a shader from the contents of the file bu

free(buf);
```

## Some Problems with the Heap

That's all well and good, but now you have a pointer to some memory that was allocated dynamically from the heap, and you must remember to free it.

```
// C
free(layer_properties);

// C++
delete[] layer_properties;

// C++ (alternative)
// Memory is freed when the std::vector goes out of
```

That allocation/free pair is what causes memory leaks: you allocate some memory with **malloc** or **new** but you forget to then use **free** or **delete**. However, if you use C++'s **std::vector** then the memory will be freed for you when the vector goes out of scope.

If you do remember to free the memory but then you accidentally try to use the pointer afterward, you'll get a segmentation fault. This sort of thing is what **Resource Acquisition Is Initialization (RAII)** and smart pointers in C++ are supposed to help with. If you allocate memory in the constructor and free it in the destructor then you don't have to worry about memory leaking. Or, if you use a smart pointer, the memory will be freed automatically when the pointer goes out of scope.

There are performance implications as well. **malloc** and **new** are written to be general-purpose and work for all cases all the time, and that generality will have some overhead. Furthermore, each call to **malloc** or **new** (which **std::vector** is doing under the hood) requires a context switch from user mode to kernel mode and then a switch back from kernel mode to user mode. These context switches may not have much of an impact when done rarely but the performance cost adds up when you're doing it many times per frame.

Finally, frequently allocating and freeing memory from the heap can cause memory fragmentation and eventual failure. As you request little chunks of memory from the OS and then free them, the heap space gets full of holes. At some point it's possible to request a chunk of memory and it to fail because the operating system can't find a contiguous chunk big enough. Even though there might be enough free space in

total, there isn't enough free space in one contiguous chunk. If that happens then the calls to **malloc** and **new** will return a null pointer and your game will crash when you attempt to use it.

These sorts of issues are more applicable to games than to other sorts of software because games need to run with high performance and they need to be able to run for many hours without crashing.

## Alternative #1: Static Arrays

You can often get away with just allocating a static array big enough to handle the largest amount of data you might ever need.

```
1     uint32_t layer_count = 0;
2     VkLayerProperties layer_properties[32];
3     vkEnumerateInstanceLayerProperties(&layer_count, NUL
4     assert(layer_count < 32);
5     vkEnumerateInstanceLayerProperties(&layer_count, lay
```

In this case we make the assumption that there will never be more than 32 layer properties and so we can just use an array allocated on the stack. We can use an **assert** to check our assumption.

This is the simplest approach but it has the side effect of wasted memory. If you know that on average you'll need eight of some item but every so often you need one thousand, then you need to have an array that can hold one thousand.

You also might need a larger array than your stack size allows which will cause a crash.

But it can sometimes still be a good choice.

## Alternative #2: Temporary Memory Allocator

This is my favorite method because of its elegance and simplicity.

You allocate a large pool of memory from the heap at startup and then you allocate from that as needed. The mem-

ory allocation happens only once, removing the costly calls to the OS, but you're still able to request arbitrary amounts of memory as if you were using **malloc** or **new**. But it uses simple pointer arithmetic and a stack which is a lot simpler than more complicated general-purpose memory allocators.

This implementation assumes there is only a single thread using the temp memory, and it doesn't consider memory alignment. You may wish to only hand out memory addresses that are aligned for SIMD access, for example.

```
typedef struct {
    void* current;
    void* end;
    int index;
    void* stack[MAX_STACK_COUNT];
} temp_mem_t;
```

We define a simple structure that is nothing more than some pointers and an index. Its general usage looks something like this:

```
// Request 16 bytes of temporary memory (once at th
temp_mem_t mem = create_temp_mem(16);

// Begin a scoped temp memory frame
begin_temp_mem(&mem);

// Request as needed
uint8_t* data0 = get_temp_mem(&mem, 4);
uint8_t* data1 = get_temp_mem(&mem, 1);
uint8_t* data2 = get_temp_mem(&mem, 6);
uint8_t* data3 = get_temp_mem(&mem, 4);

// End the scoped frame
end_temp_mem(&mem);

// Free the memory at the very end of the program
temp_mem_destroy(&mem);
```

We first need to create a **temp_mem_t** object which we can then use throughout the lifetime of our game. Before we use

any temp memory we first call **begin_temp_mem** and then when we're done we call **end_temp_mem**. At the end of our game during shutdown we can destroy it.

### Stepping Through

```
1     temp_mem_t create_temp_mem(size_t size)
2     {
3         temp_mem_t temp_mem = {0};
4         temp_mem.current = malloc(size);
5         temp_mem.end = temp_mem.current + size;
6
7         // Make sure we were able to get enough memory
8         assert(temp_mem.current);
9
10        return temp_mem;
11    }
```

Initialization is simple. We use **malloc** to request a large chunk of size **size** from the OS and we store the address of that chunk in the **current** variable. We also cache an **end** pointer which points to the end of the requested chunk. We can use the **end** pointer to check and make sure we never exceed the amount of memory that we initially allocated.

```
1     void begin_temp_mem(temp_mem_t* mem)
2     {
3         // Make sure that we haven't nested too deeply
4         assert(mem->index < MAX_STACK_COUNT);
5
6         // Save our previous memory location
7         mem->stack[mem->index] = mem->current;
8         mem->index++;
9     }
10
11    void end_temp_mem(temp_mem_t* mem)
12    {
13        // Make sure we haven't called end() more times
14        assert(mem->index > 0);
15
16        // Restore our state to the previous memory loc
17        mem->index--;
18        mem->current = mem->stack[mem->index];
19
20        // This isn't necessary but it makes things cle
21        mem->stack[mem->index] = NULL;
22    }
```

A scope begins by pushing the old **current** pointer to the stack, and a scope ends by popping a pointer from the stack back into the **current** pointer.

```
1     void* get_temp_mem(temp_mem_t* mem, size_t size)
2     {
3         void* buf = (uint8_t*)mem->current + size;
4
5         // Make sure that we aren't requesting more mem
6         assert(buf < mem->end);
7
8         // Save where we were
9         void* old = mem->current;
10        mem->current = buf;
11
12        return old;
13    }
```

Requesting new memory is a matter of moving the **current** pointer to accomodate the new size and returning the old

pointer which can now be filled out safely.

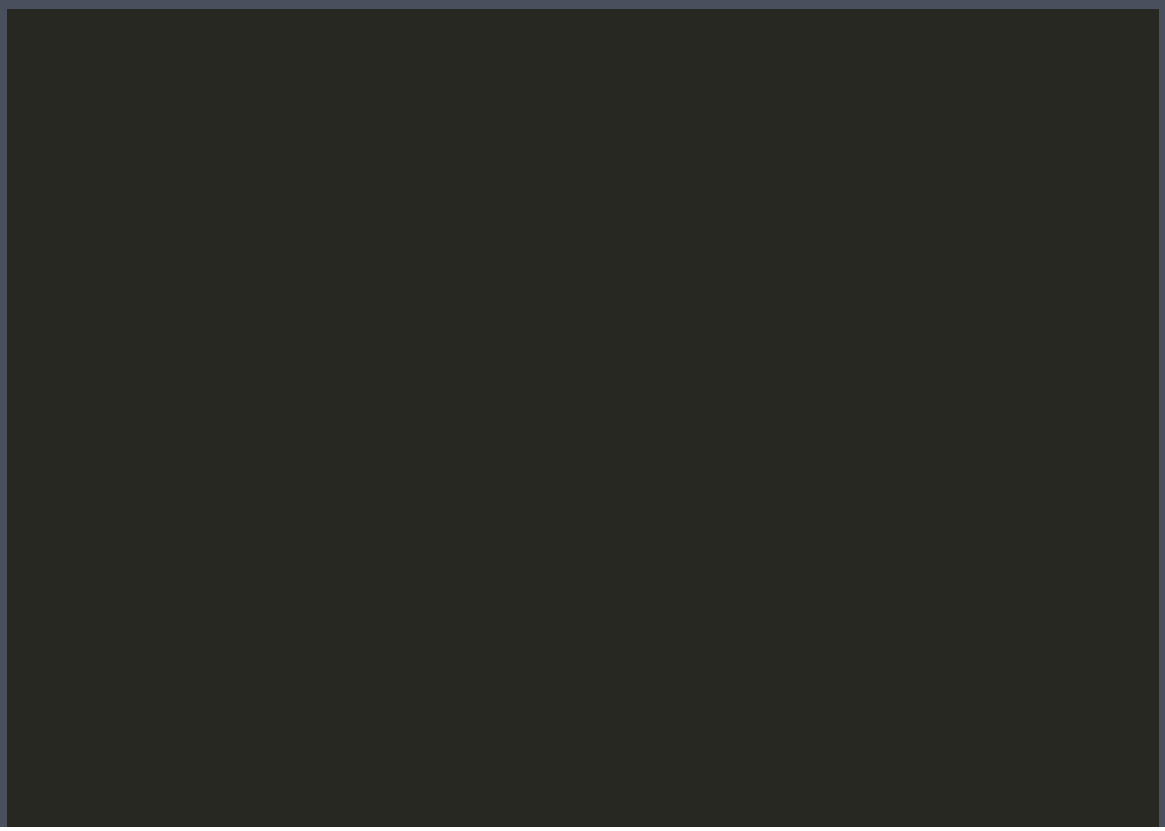
In our example, we requested a temp memory buffer of 16 bytes. In reality you'd likely want something more like a megabyte or gigabyte (depending on your application), but 16 bytes is nice for demonstration purposes. We also assume that **malloc** returns an address of 0x00, again for demonstration purposes.

At this point we can request memory from the temporary pool and it will return the **current** pointer and then increment it by the amount of bytes requested. That way any subsequent requests won't clobber the data that's already been given out.

### The Stack

What happens if you've requested some temporary memory (so you're inside of a begin/end block) and then you call a function which itself requests some temp memory? You'd clobber the other function's data. You need a way to keep track of the pointers so that you don't overwrite anything and so you can return back to where you were.

Each call to **begin_temp_mem** pushes the previous address onto the stack and each subsequent call to **end_temp_mem** pops that value off the stack into **current**.



```
1    void foo(temp_mem_t* mem)
2    {
3        /*
4        mem.index:   0
5        mem.current: 0x00
6
7        STACK
8        -------
9        0: NULL   <-- index
10       1: NULL
11       2: NULL
12       3: NULL
13       */
14
15       begin_temp_mem(mem);
16
17       /*
18       mem.index:   1
19       mem.current: 0x00
20
21       STACK
22       -------
23       0: 0x00
24       1: NULL   <-- index
25       2: NULL
26       3: NULL
27       */
28
29       // data0 = 0x00
30       uint8_t* data0 = get_temp_mem(mem, 4);
31
32       /*
33       mem.index:   1
34       mem.current: 0x04
35
36       STACK
37       -------
38       0: 0x00
39       1: NULL   <-- index
40       2: NULL
41       3: NULL
42       */
43
44       // See bar() definition below
45       bar(mem);
```

```
46
47          // data1 = 0x04
48          uint8_t* data1 = get_temp_mem(mem, 1);
49
50          /*
51          mem.index:   1
52          mem.current: 0x05
53
54          STACK
55          -------
56          0: 0x00
57          1: NULL   <-- index
58          2: NULL
59          3: NULL
60          */
61
62          end_temp_mem(mem);
63
64          /*
65          mem.index:   0
66          mem.current: 0x00
67
68          STACK
69          -------
70          0: NULL   <-- index
71          1: NULL
72          2: NULL
73          3: NULL
74          */
75      }
76
77      void bar(temp_mem_t* mem)
78      {
79          /*
80          mem.index:   1
81          mem.current: 0x04
82
83          STACK
84          -------
85          0: 0x00
86          1: NULL   <-- index
87          2: NULL
88          3: NULL
89          */
90
91          begin_temp_mem(mem);
```

```
93          /*
94          mem.index:   2
95          mem.current: 0x04
96
97          STACK
98          -------
99          0: 0x00
100         1: 0x04
101         2: NULL   <-- index
102         3: NULL
103         */
104
105         // data1 = 0x04
106         uint8_t* data0 = get_temp_mem(mem, 2);
107
108         /*
109         mem.index:   2
110         mem.current: 0x06
111
112         STACK
113         -------
114         0: 0x00
115         1: 0x04
116         2: NULL   <-- index
117         3: NULL
118         */
119
120         // data1 = 0x06
121         uint8_t* data1 = get_temp_mem(mem, 2);
122
123         /*
124         mem.index:   2
125         mem.current: 0x08
126
127         STACK
128         -------
129         0: 0x00
130         1: 0x04
131         2: NULL   <-- index
132         3: NULL
133         */
134
135         end_temp_mem(mem);
136
137         /*
138         mem.index:   1
```

```
139         mem.current: 0x04
140
141         STACK
142         -------
143         0: 0x00
144         1: NULL   <-- index
145         2: NULL
146         3: NULL
147         */
148     }
```

### Managing Scope

A flaw with this approach is that you must remember to call **end_temp_mem**. There's a cool trick we can use to create some macros that will catch that error for us.

```
1     #define BEGIN_TEMP_MEM(mem) \
2     do { \
3         goto temp_label_end; \
4     temp_label_begin: \
5         begin_temp_mem(mem);\
6     } while (0)
7
8     #define END_TEMP_MEM(mem) \
9     do {                  \
10        end_temp_mem(mem); \
11        break; \
12    temp_label_end: \
13        goto temp_label_begin; \
14    } while (0)
```

The **do while** part is just macro magic to ensure we can put our macros inside of another block. The goto labels ensure that if we use one with the other we'll have a compile-time error as **temp_label_begin** won't be defined without **BEGIN_TEMP_MEM** and **temp_label_end** won't be defined without **END_TEMP_MEM**. The only catch is that we can only have one temp memory region per function.

We can then use them like this:

```
```

```c
int main(void)
{
    temp_mem_t mem = create_temp_mem(16);

    BEGIN_TEMP_MEM(&mem);

    uint8_t* data0 = get_temp_mem(&mem, 4);
    uint8_t* data1 = get_temp_mem(&mem, 1);
    uint8_t* data2 = get_temp_mem(&mem, 6);
    uint8_t* data3 = get_temp_mem(&mem, 4);

    END_TEMP_MEM(&mem);

    temp_mem_destroy(&mem);

    return 0;
}
```

If using C++ we can help prevent this by creating a separate class that calls **begin_temp_mem** in its constructor and **end_temp_mem** in its destructor. Then we just instantiate the class at the beginning of our tempory memory scope and when falling out of scope the destructor will be called automatically which will call **end_temp_mem** for us.

Personally I prefer the C approach of seeing the explicit calls to **begin_temp_mem** and **end_temp_mem**, but I recognize the danger of forgetting to call one or the other.

### A Global Object

You might also notice that passing around a **temp_mem_t** object to every function is a pain. Similar to something like a logger, it would be better to create some global object somewhere that everyone can access. You might disagree with this philosophically (“globals are bad”) but it works. You could also create a static singleton class that holds a reference to the temporary memory but not everyone is a fan of singletons either.

You can handle it however you want. Since I’m the only person working on my personal projects, I opt for a single **temp_mem_t** that is static to my **memory.c** file. I can just call the relevant functions without needing to worry about the actual **temp_mem_t** object.

## In Practice

Here is how I would do it if you just wanted a single **temp_mem_t** that is static to the allocator file.

```c
/* memory.c */

#define TEMP_MEM_SIZE (1U * 1024U * 1024U)
#define TEMP_MEM_STACK_LENGTH (8)

static struct {
    void* current;
    void* end;
    size_t index;
    void* stack[TEMP_MEM_STACK_LENGTH];
} s_temp_mem;

void allocator_startup(void)
{
    s_temp_mem.current = malloc(TEMP_MEM_SIZE);
    s_temp_mem.end = (uint8_t*)s_temp_mem.current +

    assert(s_temp_mem.current);
}

void allocator_shutdown(void)
{
    free(s_temp_mem.current);
    s_temp_mem.current = NULL;
    s_temp_mem.end = NULL;
}

void begin_temp_mem(void)
{
    assert(s_temp_mem.index < TEMP_MEM_STACK_LENGTH

    s_temp_mem.stack[s_temp_mem.index] = s_temp_mem
    s_temp_mem.index++;
}

void end_temp_mem(void)
{
    assert(s_temp_mem.index > 0);

    s_temp_mem.index--;
    s_temp_mem.current = s_temp_mem.stack[s_temp_me
    s_temp_mem.stack[s_temp_mem.index] = NULL;
}

void* get_temp_mem(size_t size)
```

```
46      {
47          void* buf = (uint8_t*)s_temp_mem.current + size
48
49          assert(buf < s_temp_mem.end);
50
51          void* old = s_temp_mem.current;
52          s_temp_mem.current = buf;
53
54          return old;
55      }
56
57      #undef TEMP_MEM_SIZE
58      #undef TEMP_MEM_STACK_LENGTH
```

## Conclusion

Hopefully you find the concept of a large chunk of temporary memory to be as compelling and elegant as I do.

There is another side to the dynamic memory coin which is when you need memory from the heap but with a lifetime that is larger than a single scope. That requires a custom memory allocator which again allocates a large chunk during game startup but it's responsible for handing out memory, keeping a free list, rearranging to remove holes, etc. That's a topic for a different day.

## Further Reading

- **Game Engine Architecture**
- **Stack Overflow**

Last Edited: Dec 20, 2022